P10076US

**IN THE UNITED STATES PATENT AND TRADEMARK OFFICE**

**APPLICATION PAPERS**

**OF**

**EDWARD COLLES NEVILL**

**FOR**

# INTERCALLING BETWEEN NATIVE AND NON-NATIVE INSTRUCTION SETS

## BACKGROUND OF THE INVENTION

### Field of the Invention

This invention relates to the field of data processing. More particularly, this invention relates to data processing systems supporting both native instructions and interpreted non-native instructions and within which it is desired to make subroutine calls from native code to a non-native subroutine.

### Description of the Prior Art

It is a well known computer programming technique to utilise subroutine calls within a program structure. Such subroutine calls allow frequently used code segments to be re-used from different points within a computer program. When a computer program reaches a point at which it is desired to start execution of a subroutine, a call instruction will be executed to redirect processing to the start of the subroutine. At the end of the subroutine, a return instruction will be executed that returns processing control to the address within the program code immediately following the point from which the subroutine call was made.

It is known to provide computer program operations utilising a mix of native instructions that may be directly executed by a processor core and non-native instructions that require interpretation into the form of native instructions prior to execution. Typically, native instructions are more rapid to execute but have a lower code density and are more difficult to program than non-native instructions.

Figure 14 of the accompanying drawings illustrates a known technique whereby a non-native caller routine may make a subroutine call to a non-native subroutine. The call is made by an invoke_static instruction that passes processing control to the start of the corresponding subroutine. At the end of the subroutine, a return instruction is executed that returns processing to the non-native instruction immediately following the invoke_static instruction.

Figure 15 of the accompanying drawings illustrates a native caller program used to call a native subroutine. In this example, the caller native program and the subroutine native program are written in ARM object code form (using the native instructions of an ARM processor as produced by ARM Limited of Cambridge, England). The caller program issues a native call instruction BL<address> that passes control to the start of the native subroutine program. The BL<address> also stores a return address in a register of the processor core (in this example the link register lr) that points to the address following the subroutine call instruction. At the end of the native subroutine, a return instruction in the form of a MOVS pc lr native instruction serves to move the stored return address from the link register lr into the program counter register pc thereby directing processing to resume at the native instruction following the original calling instruction.

Figure 16 of the accompanying drawings illustrates a common non-native subroutine that may be called by either native calling code or non-native calling code. Since the non-native subroutine is executed in a non-native mode, when the subroutine has completed, it is required to determine whether a return is being made to a non-native calling program, in which case a mode change is not required, or alternative to a native calling program, in which case a mode change is required. In order to achieve this, the return instruction at the end of the non-native subroutine has to test to determine whether or not a return is being made to a native calling program or a non-native calling program before the return is actually made. This slows down processing. Given below is an example of some ARM object code instructions that may be interpreted from the non-native return instruction at the end of the subroutine of Figure 16 in order to carry out the appropriate determination and return to the native calling program or the non-native calling program as necessary:

```
do_return
                LDR         Rtmp, [Rframe, #previous_frame]
                CMP         Rtmp, #0
                BEQ         return_to_native
return_to_java
                Code to perform return to non-native Java code
return_to_native
                Code to perform return to native ARM or Thumb code
```

In this example code the return instruction is a Java bytecode that has been interpreted by a Java software interpreter into ARM object code that serves to determine whether or not the previous frame value from which the call was made was #0. If the previous frame value was #0, then this indicates that the calling program was a native program and a return to a native program mode (i.e. ARM object code) is made. If the previous frame value is not #0, then the calling program was a non-native program and a return to a non-native mode (i.e. Java) is made.

It will be appreciated that the extra processing needed each time a return is made to determine whether or not the calling program was a native calling program or a non-native calling program is a performance disadvantage. This is particularly the case since in a large majority of cases non-native subroutines are called from non-native calling programs and native subroutines are called from native calling programs with comparatively little intercalling between native and non-native routines. Thus, the checking in an overwhelming majority of cases produces a result indicating the mode is unchanged but must nevertheless take place on each return instruction in order to cater for the relatively rare cases of an intercall between native and non-native code.

Figure 17 of the accompanying drawings illustrates a known system whereby a non-native veneer subroutine is called from calling native code as an indirect technique for triggering execution of a main non-native subroutine. The use of such veneer subroutines is a known technique for efficient programming.

Examples of known systems for translation between instruction sets and other background information may be found in the following: US-A-5,805,895; US-A-3,955,180; US-A-5,970,242; US-A-5,619,665; US-A-5,826,089; US-A-5,925,123; US-A-5,875,336; US-A-5,937,193; US-A-5,953,520; US-A-6,021,469; US-A-5,568,646; US-A-5,758,115; UA-A-5,367,685; IBM Technical Disclosure Bulletin, March 1988, pp308-309, "System/370 Emulator Assist Processor For a Reduced Instruction Set Computer"; IBM Technical Disclosure Bulletin, July 1986, pp548-549, "Full Function Series/1 Instruction Set Emulator"; IBM Technical Disclosure Bulletin, March 1994, pp605-606, "Real-Time CISC Architecture HW Emulator On A RISC Processor"; IBM Technical Disclosure Bulletin, March 1998, p272, "Performance Improvement Using An EMULATION Control Block"; IBM Technical Disclosure Bulletin, January 1995, pp537-540, "Fast Instruction Decode For Code Emulation on Reduced Instruction Set Computer/Cycles Systems"; IBM Technical Disclosure Bulletin, February 1993, pp231-234, "High Performance Dual Architecture Processor"; IBM Technical Disclosure Bulletin, August 1989, pp40-43, "System/370 I/O Channel Program Channel Command Word Prefetch"; IBM Technical Disclosure Bulletin, June 1985, pp305-306, "Fully Microcode-Controlled Emulation Architecture"; IBM Technical Disclosure Bulletin, March 1972, pp3074-3076, "Op Code and Status Handling For Emulation"; IBM Technical Disclosure Bulletin, August 1982, pp954-956, "On-Chip Microcoding of a Microprocessor With Most Frequently Used Instructions of Large System and Primitives Suitable for Coding Remaining Instructions"; IBM Technical Disclosure Bulletin, April 1983, pp5576-5577, "Emulation Instruction"; the book ARM System Architecture by S Furber; the book Computer Architecture: A Quantitative Approach by Hennessy and Patterson; and the book The Java Virtual Machine Specification by Tim Lindholm and Frank Yellin 1st and 2nd editions.

## SUMMARY OF THE INVENTION

Viewed from one aspect the present invention provides apparatus for processing data, said apparatus comprising:

(i) a processor core operable to execute native instructions of a native instruction set; and

(ii) an instruction translator operable to interpret non-native instructions of a non-native instruction set into native instructions for execution by said processor core; wherein

(iii) said instruction translator is responsive to a return to non-native instruction of said non-native instruction set to return processing to a non-native instruction; and

(iv) said instruction translator is responsive to a return to native instruction of said non-native instruction set to return processing to a native instruction.

The present invention recognises the above described problems of intercalling between native and non-native instructions together with the inefficiencies of checking at each return instruction whether or not the call was made from native or non-native code. In order to reduce these problems the invention provides separate and distinct instructions in the form of a return to non-native instruction and a return to native instruction whereby processing is respectively returned to a non-native instruction and a native instruction. In this way, flexibility to return either to a native instruction or to a non-native instruction may be provided without having to support the additional overhead of checking for the nature of the calling program each time a return instruction is encountered. Instead, specific types of return instruction are provided that effectively code into the program the nature of the return to be made.

It will be appreciated that the instruction translator could take many different forms. In particular, the instruction translator may be a hardware based instruction translator, a software based interpreter or a mixture of the two.

The interpreted instructions could take many different forms. However, the invention is particularly well suited to embodiments in which the non-native instructions are Java Virtual Machine Instructions. Such Java Virtual Machine Instructions make frequent use of subroutine calls and also are readily mixed with native instructions in high performance efficient embodiments.

Whilst it is possible that separate subroutines may be individually provided with their own return instructions to respectively provide for calls from native and non-native calling programs, in preferred embodiments of the invention a non-native subroutine is called from native code via a non-native veneer subroutine, such that, upon completion of said non-native subroutine, a return to non-native instruction can be used to return processing to said non-native veneer subroutine with a return to native instruction within said non-native veneer subroutine serving to return processing to said native code.

The use of a non-native veneer subroutine between native code and a non-native subroutine allows the non-native subroutine to be re-used with only the small veneer non-native subroutine needing to be separately provided to enable calls to be made from native code as opposed to the standard return at the end of the subroutine that is a non-native return.

The additional memory consumed by the need to provide veneer subroutines may be advantageously reduced in embodiments of the invention in which the veneer subroutines are dynamically created when a non-native subroutine is called from native code. A particularly preferred way of achieving this is to create the non-native veneer subroutine within a stack memory area of the native code operational mode such that the normal system resources can be used to provide storage for this non-native veneer subroutine and the nesting of subroutines may be conveniently supported.

In preferred sophisticated embodiments of the invention a plurality of types of return to non-native instruction are provided which can return different types of return value, such as a 32-bit integer, a 64-bit integer, an object reference, a single precision floating point value, a double precision floating point value or a void value having no value.

Whilst in some embodiments only a single return to native instruction may be provided so as to preserve opcode space within the interpreted instruction set, within other embodiments it may be preferred to similarly provide a plurality of return to native instructions within the interpreted instruction set.

Viewed from another aspect the present invention provides a method of processing data, said method comprising the steps of:

(i) executing native instructions of a native instruction set using a processor core; and

(ii) interpreting non-native instructions of a non-native instruction set into native instructions for execution by said processor core; wherein

(iii) in response to a return to non-native instruction of said non-native instruction set, returning processing to a non-native instruction; and

(iv) in response to a return to native instruction of said non-native instruction set, returning processing to a native instruction.

The above, and other objects, features and advantages of this invention will be apparent from the following detailed description of illustrative embodiments which is to be read in connection with the accompanying drawings.

## BRIEF DESCRIPTION OF THE DRAWINGS

Figures 1 and 2 schematically represent example instruction pipeline arrangements;

Figure 3 illustrates in more detail a fetch stage arrangement;

Figure 4 schematically illustrates the reading of variable length non-native instructions from within buffered instruction words within the fetch stage;

Figure 5 schematically illustrates a data processing system for executing both processor core native instructions and instructions requiring translation;

Figure 6 schematically illustrates, for a sequence of example instructions and states the contents of the registers used for stack operand storage, the mapping states and the relationship between instructions requiring translation and native instructions;

Figure 7 schematically illustrates the execution of a non-native instruction as a sequence of native instructions;

Figure 8 is a flow diagram illustrating the way in which the instruction translator may operate in a manner that preserves interrupt latency for translated instructions;.

Figure 9 schematically illustrates the translation of Java bytecodes into ARM opcodes using hardware and software techniques;

Figure 10 schematically illustrates the flow of control between a hardware based translator, a software based interpreter and software based scheduling;

Figures 11 and 12 illustrate another way of controlling scheduling operations using a timer based approach;

Figure 13 is a signal diagram illustrating the signals controlling the operation of the circuit of Figure 12;

Figures 14 to 17 schematically illustrate known techniques for calling between calling programs and subroutines in native and non-native programs;

Figure 18 schematically illustrates a first embodiment for providing subroutine return instructions that explicitly return either to native code or non-native code;

Figure 19 schematically illustrates the use of a non-native veneer subroutine between a native calling program and non-native subroutine;

Figure 20 schematically illustrates a data processing system which dynamically creates a non-native veneer subroutine as used in Figure 19; and

Figure 21 is a flow diagram schêmatically illustrating the processing performed in the system of Figure 20.

## DESCRIPTION OF THE PREFERRED EMBODIMENTS

Figure 1 shows a first example instruction pipeline 30 of a type suitable for use in an ARM processor based system. The instruction pipeline 30 includes a fetch stage 32, a native instruction (ARM/Thumb instructions) decode stage 34, an execute stage 36, a memory access stage 38 and a write back stage 40. The execute stage 36, the memory access stage 38 and the write back stage 40 are substantially conventional. Downstream of the fetch stage 32, and upstream of the native instruction decode stage 34, there is provided an instruction translator stage 42. The instruction translator stage 42 is a finite state machine that translates Java bytecode instructions of a variable length into native ARM instructions. The instruction translator stage 42 is capable of multi-step operation whereby a single Java bytecode instruction may generate a sequence of ARM instructions that are fed along the remainder of the instruction pipeline 30 to perform the operation specified by the Java bytecode instruction. Simple Java bytecode instructions may required only a single ARM instruction to perform their operation, whereas more complicated Java bytecode instructions, or in circumstances where the surrounding system state so dictates, several ARM instructions may be needed to provide the operation specified by the Java bytecode instruction. This multi-step operation takes place downstream of the fetch stage 32 and accordingly power is not expended upon fetching multiple translated ARM instructions or Java bytecodes from a memory system. The Java bytecode instructions are stored within the memory system in a conventional manner such that additional constraints are not provided upon the memory system in order to support the Java bytecode translation operation.

As illustrated, the instruction translator stage 42 is provided with a bypass path. When not operating in an instruction translating mode, the instruction pipeline 30 may bypass the instruction translator stage 42 and operate in an essentially unaltered manner to provide decoding of native instructions.

In the instruction pipeline 30, the instruction translator stage 42 is illustrated as generating translator output signals that fully represent corresponding ARM instructions and are passed via a multiplexer to the native instruction decoder 34. The instruction translator 42 also generates some extra control signals that may be passed to the native instruction decoder 34. Bit space constraints within the native instruction encoding may impose limitations upon the range of operands that may be specified by native instructions. These limitations are not necessarily shared by the non-native instructions. Extra control signals are provided to pass additional instruction specifying signals derived from the non-native instructions that would not be possible to specify within native instructions stored within memory. As an example, a native instruction may only provide a relatively low number of bits for use as an immediate operand field within a native instruction, whereas the non-native instruction may allow an extended range and this can be exploited by using the extra control signals to pass the extended portion of the immediate operand to the native instruction decoder 34 outside of the translated native instruction that is also passed to the native instruction decoder 34.

Figure 2 illustrates a further instruction pipeline 44. In this example, the system is provided with two native instruction decoders 46, 48 as well as a non-native instruction decoder 50. The non-native instruction decoder 50 is constrained in the operations it can specify by the execute stage 52, the memory stage 54 and the write back stage 56 that are provided to support the native instructions. Accordingly, the non-native instruction decoder 50 must effectively translate the non-native instructions into native operations (which may be a single native operation or a sequence of native operations) and then supply appropriate control signals to the execute stage 52 to carry out these one or more native operations. It will be appreciated that in this example the non-native instruction decoder does not produce signals that form a native instruction, but rather provides control signals that specify native instruction (or extended native instruction) operations. The control signals generated may not match the control signals generated by the native instruction decoders 46, 48.

In operation, an instruction fetched by the fetch stage 58 is selectively supplied to one of the instruction decoders 46, 48 or 50 in dependence upon the particular processing mode using the illustrated demultiplexer.

Figure 3 schematically illustrates the fetch stage of an instruction pipeline in more detail. Fetching logic 60 fetches fixed length instruction words from a memory system and supplies these to an instruction word buffer 62. The instruction word buffer 62 is a swing buffer having two sides such that it may store both a current instruction word and a next instruction word. Whenever the current instruction word has been fully decoded and decoding has progressed onto the next instruction word, then the fetch logic 60 serves to replace the previous current instruction word with the next instruction word to be fetched from memory, i.e. each side of the swing buffer will increment by two in an interleaved fashion the instruction words that they successively store.

In the example illustrated, the maximum instruction length of a Java bytecode instruction is three bytes. Accordingly, three multiplexers are provided that enable any three neighbouring bytes within either side of the word buffer 62 to be selected and supplied to the instruction translator 64. The word buffer 62 and the instruction translator 64 are also provided with a bypass path 66 for use when native instructions are being fetched and decoded.

It will be seen that each instruction word is fetched from memory once and stored within the word buffer 62. A single instruction word may have multiple Java bytecodes read from it as the instruction translator 64 performs the translation of Java bytecodes into ARM instructions. Variable length translated sequences of native instructions may be generated without requiring multiple memory system reads and without consuming memory resource or imposing other constraints upon the memory system as the instruction translation operations are confined within the instruction pipeline.

A program counter value is associated with each Java bytecode currently being translated. This program counter value is passed along the stages of the pipeline such that each stage is able, if necessary, to use the information regarding the particular Java bytecode it is processing. The program counter value for a Java bytecode that translates into a sequence of a plurality of ARM instruction operations is not incremented until the final ARM instruction operation within that sequence starts to be executed. Keeping the program counter value in a manner that continues to directly point to the instruction within the memory that is being executed advantageously simplifies other aspects of the system, such as debugging and branch target calculation.

Figure 4 schematically illustrates the reading of variable length Java bytecode instructions from the instruction buffer 62. At the first stage a Java bytecode instruction having a length of one is read and decoded. The next stage is a Java bytecode instruction that is three bytes in length and spans between two adjacent instruction words that have been fetched from the memory. Both of these instruction words are present within the instruction buffer 62 and so instruction decoding and processing is not delayed by this spanning of a variable length instruction between instruction words fetched. Once the three Java bytecodes have been read from the instruction buffer 62, the refill of the earlier fetched of the instruction words may commence as subsequent processing will continue with decoding of Java bytecodes from the following instruction word which is already present.

The final stage illustrated in Figure 4 illustrates a second three bytecode instruction being read. This again spans between instruction words. If the preceding instruction word has not yet completed its refill, then reading of the instruction may be delayed by a pipeline stall until the appropriate instruction word has been stored into the instruction buffer 62. In some embodiments the timings may be such that the pipeline never stalls due to this type of behaviour. It will be appreciated that the particular example is a relatively infrequent occurrence as most Java bytecodes are shorter than the examples illustrated and accordingly two successive decodes that both span between instruction words is relatively uncommon. A valid signal may be associated with each of the instruction words within the instruction buffer 62 in a manner that is able to signal whether or not the instruction word has appropriately been refilled before a Java bytecode has been read from it.

Figure 5 shows a data processing system 102 including a processor core 104 and a register bank 106. An instruction translator 108 is provided within the instruction path to translate Java Virtual Machine instructions to native ARM instructions (or control signals corresponding thereto) that may then be supplied to the processor core 104. The instruction translator 108 may be bypassed when native ARM instructions are being fetched from the addressable memory. The addressable memory may be a memory system such as a cache memory with further off-chip RAM memory. Providing the instruction translator 108 downstream of the memory system, and particularly the cache memory, allows efficient use to be made of the storage capacity of the memory system since dense instructions that require translation may be stored within the memory system and only expanded into native instructions immediately prior to being passed to the processor core 104.

The register bank 106 in this example contains sixteen general purpose 32-bit registers, of which four are allocated for use in storing stack operands, i.e. the set of registers for storing stack operands is registers R0, R1, R2 and R3.

The set of registers may be empty, partly filled with stack operands or completely filled with stack operands. The particular register that currently holds the top of stack operand may be any of the registers within the set of registers. It will thus be appreciated that the instruction translator may be in any one of seventeen different mapping states corresponding to one state when all of the registers are empty and four groups of four states each corresponding to a respective different number of stack operands being held within the set of registers and with a different register holding the top of stack operand. Table 1 illustrates the seventeen different states of the state mapping for the instruction translator 108. It will be appreciated that with a different number of registers allocated for stack operand storage, or as a result of constraints that a particular processor core may have in the way it can manipulate data values held within registers, the mapping states can very considerably depending upon the particular implementation and Table 1 is only given as an example of one particular implementation.

| | | | |
|---|---|---|---|
| STATE 00000 | | | |
| R0 = EMPTY | | | |
| R1 = EMPTY | | | |
| R2 = EMPTY | | | |
| R3 = EMPTY | | | |
| STATE 00100 | STATE 01000 | STATE 01100 | STATE 10000 |
| R0 = TOS | R0 = TOS | R0 = TOS | R0 = TOS |
| R1 = EMPTY | R1 = EMPTY | R1 = EMPTY | R1 = TOS-3 |
| R2 = EMPTY | R2 = EMPTY | R2 = TOS-2 | R2 = TOS-2 |
| R3 = EMPTY | R3 = TOS-1 | R3 = TOS-1 | R3 = TOS-1 |
| STATE 00101 | STATE 01001 | STATE 01101 | STATE 10001 |
| R0 = EMPTY | R0 = TOS-1 | R0 = TOS-1 | R0 = TOS-1 |
| R1 = TOS | R1 = TOS | R1 = TOS | R1 = TOS |
| R2 = EMPTY | R2 = EMPTY | R2 = EMPTY | R2 = TOS-3 |
| R3 = EMPTY | R3 = EMPTY | R3 = TOS-2 | R3 = TOS-2 |
| STATE 00110 | STATE 01010 | STATE 01110 | STATE 10010 |
| R0 = EMPTY | R0 = EMPTY | R0 = TOS-2 | R0 = TOS-2 |
| R1 = EMPTY | R1 = TOS-1 | R1 = TOS-1 | R1 = TOS-1 |
| R2 = TOS | R2 = TOS | R2 = TOS | R2 = TOS |
| R3 = EMPTY | R3 = EMPTY | R3 = EMPTY | R3 = TOS-3 |
| STATE 00111 | STATE 01011 | STATE 01111 | STATE 10011 |
| R0 = EMPTY | R0 = EMPTY | R0 = EMPTY | R0 = TOS-3 |
| R1 = EMPTY | R1 = EMPTY | R1 = TOS-2 | R1 = TOS-2 |

| | | | |
|---|---|---|---|
| R2 = EMPTY<br>R3 = TOS | R2 = TOS-1<br>R3 = TOS | R2 = TOS-1<br>R3 = TOS | R2 = TOS-1<br>R3 = TOS |

TABLE 1

Within Table 1 it may be observed that the first three bits of the state value indicate the number of non-empty registers within the set of registers. The final two bits of the state value indicate the register number of the register holding the top of stack operand. In this way, the state value may be readily used to control the operation of a hardware translator or a software translator to take account of the currently occupancy of the set of registers and the current position of the top of stack operand.

As illustrated in Figure 5 a stream of Java bytecodes J1, J2, J3 is fed to the instruction translator 108 from the addressable memory system. The instruction translator 108 then outputs a stream of ARM instructions (or equivalent control signals, possibly extended) dependent upon the input Java bytecodes and the instantaneous mapping state of the instruction translator 8, as well as other variables. The example illustrated shows Java bytecode J1 being mapped to ARM instructions $A^1 1$ and $A^1 2$. Java bytecode J2 maps to ARM instructions $A^2 1$, $A^2 2$ and $A^2 3$. Finally, Java bytecode J3 maps to ARM instruction $A^3 1$. Each of the Java bytecodes may require one or more stack operands as inputs and may produce one or more stack operands as an output. Given that the processor core 104 in this example is an ARM processor core having a load/store architecture whereby only data values held within registers may be manipulated, the instruction translator 108 is arranged to generate ARM instructions that, as necessary, fetch any required stack operands into the set of registers before they are manipulated or store to addressable memory any currently held stack operands within the set of registers to make room for result stack operands that may be generated. It will be appreciated that each Java bytecode may be considered as having an associated "require full" value indicating the number of stack operands that must be present within the set of registers prior to its execution together with a "require empty" value indicating the number of empty registers within the set of registers that must be available prior to execution of the ARM instructions representing the Java opcode.

Table 2 illustrates the relationship between initial mapping state values, require full values, final state values and associated ARM instructions. The initial state values and the final state values correspond to the mapping states illustrated in Table 1. The instruction translator 108 determines a require full value associated with the particular Java bytecode (opcode) it is translating. The instruction translator (108), in dependence upon the initial mapping state that it has, determines whether or not more stack operands need to be loaded into the set of registers prior to executing the Java bytecode. Table 1 shows the initial states together with tests applied to the require full value of the Java bytecode that are together applied to determine whether a stack operand needs to be loaded into the set of registers using an associated ARM instruction (an LDR instruction) as well as the final mapping state that will be adopted after such a stack cache load operation. In practice, if more than one stack operand needs to be loaded into the set of registers prior to execution of the Java bytecode, then multiple mapping state transitions will occur, each with an associated ARM instruction loading a stack operand into one of the registers of the set of registers. In different embodiments it may be possible to load multiple stack operands in a single state transition and accordingly make mapping state changes beyond those illustrated in Table 2.

| INITIAL STATE | REQUIRE FULL | FINAL STATE | ACTIONS |
|---|---|---|---|
| 00000 | >0 | 00100 | LDR R0, [Rstack, #-4]! |
| 00100 | >1 | 01000 | LDR R3, [Rstack, #-4]! |
| 01001 | >2 | 01101 | LDR R3, [Rstack, #-4]! |
| 01110 | >3 | 10010 | LDR R3, [Rstack, #-4]! |
| 01111 | >3 | 10011 | LDR R0, [Rstack, #-4]! |
| 01100 | >3 | 10000 | LDR R1, [Rstack, #-4]! |
| 01101 | >3 | 10001 | LDR R2, [Rstack, #-4]! |
| 01010 | >2 | 01110 | LDR R0, [Rstack, #-4]! |
| 01011 | >2 | 01111 | LDR R1, [Rstack, #-4]! |
| 01000 | >2 | 01100 | LDR R2, [Rstack, #-4]! |
| 00110 | >1 | 01010 | LDR R1, [Rstack, #-4]! |
| 00111 | >1 | 01011 | LDR R2, [Rstack, #-4]! |
| 00101 | >1 | 01001 | LDR R0, [Rstack, #-4]! |

**TABLE 2**

As will be seen from Table 2, a new stack operand loaded into the set of registers storing stack operands will form a new top of stack operand and this will be loaded into a particular one of the registers within the set of registers depending upon the initial state.

Table 3 in a similar manner illustrates the relationship between initial state, require empty value, final state and an associated ARM instruction for emptying a register within the set of registers to move between the initial state and the final state if the require empty value of a particular Java bytecode indicates that it is necessary given the initial state before the Java bytecode is executed. The particular register values stored off to the addressable memory with an STR instruction will vary depending upon which of the registers is the current top of stack operand.

| INITIAL STATE | REQUIRE EMPTY | FINAL STATE | ACTIONS |
|---|---|---|---|
| 00100 | >3 | 00000 | STR R0, [Rstack], #4 |
| 01001 | >2 | 00101 | STR R0, [Rstack], #4 |
| 01110 | >1 | 01010 | STR R0, [Rstack], #4 |
| 10011 | >0 | 01111 | STR R0, [Rstack], #4 |
| 10000 | >0 | 01100 | STR R1, [Rstack], #4 |
| 10001 | >0 | 01101 | STR R2, [Rstack], #4 |
| 10010 | >0 | 01110 | STR R3, [Rstack], #4 |
| 01111 | >1 | 01011 | STR R1, [Rstack], #4 |
| 01100 | >1 | 01000 | STR R2, [Rstack], #4 |
| 01101 | >1 | 01001 | STR R3, [Rstack], #4 |
| 01010 | >2 | 00110 | STR R1, [Rstack], #4 |
| 01011 | >2 | 00111 | STR R2, [Rstack], #4 |
| 01000 | >2 | 00100 | STR R3, [Rstack], #4 |
| 00110 | >3 | 00000 | STR R2, [Rstack], #4 |
| 00111 | >3 | 00000 | STR R3, [Rstack], #4 |
| 00101 | >3 | 00000 | STR R1, [Rstack], #4 |

**TABLE 3**

It will be appreciated that in the above described example system the require full and require empty conditions are mutually exclusive, that is to say only one of the require full or require empty conditions can be true at any given time for a particular Java bytecode which the instruction translator is attempting to translate. The instruction templates used by the instruction translator 108 together with the instructions it is chosen to support with the hardware instruction translator 108 are selected such that this mutually exclusive requirement may be met. If this requirement were not in place, then the situation could arise in which a particular Java bytecode required a number of input stack operands to be present within the set of registers that would not allow sufficient empty registers to be available after execution of the instruction representing the Java bytecode to allow the results of the execution to be held within the registers as required.

It will be appreciated that a given Java bytecode will have an overall nett stack action representing the balance between the number of stack operands consumed and the number of stack operands generated upon execution of that Java bytecode. Since the number of stack operands consumed is a requirement prior to execution and the number of stack operands generated is a requirement after execution, the require full and require empty values associated with each Java bytecode must be satisfied prior to execution of that bytecode even if the nett overall action would in itself be met. Table 4 illustrates the relationship between an initial state, an overall stack action, a final state and a change in register use and relative position of the top of stack operand (TOS). It may be that one or more of the state transitions illustrated in Table 2 or Table 3 need to be carried out prior to carrying out the state transitions illustrated in Table 4 in order to establish the preconditions for a given Java bytecode depending on the require full and require empty values of the Java bytecode.

| INITIAL STATE | STACK ACTION | FINAL STATE | ACTIONS |
|---|---|---|---|
| 00000 | +1 | 00101 | R1 <- TOS |
| 00000 | +2 | 01010 | R1 <- TOS-1, R2 <- TOS |
| 00000 | +3 | 01111 | R1 <- TOS-2, R2 <- TOS-1, R3 <- TOS |
| 00000 | +4 | 10000 | R0 <- TOS, R1 <- TOS-3, R2 <- TOS-2, R3 <- TOS-1 |
| 00100 | +1 | 01001 | R1 <- TOS |
| 00100 | +2 | 01110 | R1 <- TOS-1, R2 <- TOS |
| 00100 | +3 | 10011 | R1 <- TOS-2, R2 <- TOS-1, R3 <- TOS |
| 00100 | -1 | 00000 | R0 <- EMPTY |
| 01001 | +1 | 01110 | R2 <- TOS |
| 01001 | +2 | 10011 | R2 <- TOS-1, R3 <- TOS |
| 01001 | -1 | 00100 | R1 <- EMPTY |
| 01001 | -2 | 00000 | R0 <- EMPTY, R1 <- EMPTY |
| 01110 | +1 | 10011 | R3 <- TOS |
| 01110 | -1 | 01001 | R2 <- EMPTY |
| 01110 | -2 | 00100 | R1 <- EMPTY, R2 <- EMPTY |
| 01110 | -3 | 00000 | R0 <- EMPTY, R1 <- EMPTY, R2 <- EMPTY |
| 10011 | -1 | 01110 | R3 <- EMPTY |
| 10011 | -2 | 01001 | R2 <- EMPTY, R3 <- EMPTY |
| 10011 | -3 | 00100 | R1 <- EMPTY, R2 <- EMPTY, R3 <- EMPTY |

```
10011       -4      00000   RO <- EMPTY, R1 <- EMPTY, R2 <- EMPTY,   R3
<-
                                                                     EMP
                                                              TY

10000       -1      01111   RO <- EMPTY
10000       -2      01010   RO <- EMPTY, R3 <- EMPTY
10000       -3      00101   RO <- EMPTY, R2 <- EMPTY, R3 <- EMPTY
10000       -4      00000   RO <- EMPTY, R1 <- EMPTY, R2 <- EMPTY,   R3
<-
                                                                     EMP
                                                              TY
```

| | | | |
|---|---|---|---|
| 10001 | -1 | 01100 | R1 <- EMPTY |
| 10001 | -2 | 01011 | R0 <- EMPTY, R1 <- EMPTY |
| 10001 | -3 | 00110 | R0 <- EMPTY, R1 <- EMPTY, R3 <- EMPTY |
| 10001 | -4 | 00000 | R0 <- EMPTY, R1 <- EMPTY, R2 <- EMPTY, R3 <- EMPTY |
| 10010 | -1 | 01101 | R2 <- EMPTY |
| 10010 | -2 | 01000 | R1 <- EMPTY, R2 <- EMPTY |
| 10010 | -3 | 00111 | R0 <- EMPTY, R1 <- EMPTY, R2 <- EMPTY |
| 10010 | -4 | 00000 | R0 <- EMPTY, R1 <- EMPTY, R2 <- EMPTY, R3 <- EMPTY |
| 01111 | +1 | 10000 | R0 <- TOS |
| 01111 | -1 | 01010 | R3 <- EMPTY |
| 01111 | -2 | 00101 | R2 <- EMPTY, R3 <- EMPTY |
| 01111 | -3 | 00000 | R1 <- EMPTY, R2 <- EMPTY, R3 <- EMPTY |
| 01100 | +1 | 10001 | R1 <- TOS |
| 01100 | -1 | 01011 | R0 <- EMPTY |
| 01100 | -2 | 00110 | R0 <- EMPTY, R3 <- EMPTY |
| 01100 | -3 | 00000 | R0 <- EMPTY, R2 <- EMPTY, R3 <- EMPTY |
| 01101 | +1 | 10010 | R2 <- TOS |
| 01101 | -1 | 01000 | R1 <- EMPTY |
| 01101 | -2 | 00111 | R0 <- EMPTY, R1 <- EMPTY |
| 01101 | -3 | 00000 | R0 <- EMPTY, R1 <- EMPTY, R3 <- EMPTY |
| 01010 | +1 | 01111 | R3 <- TOS |
| 01010 | +2 | 10000 | R3 <- TOS-1, R0 <- TOS |
| 01010 | -1 | 00101 | R2 <- EMPTY |
| 01010 | -2 | 00000 | R1 <- EMPTY, R2 <- EMPTY |
| 01011 | +1 | 01100 | R0 <- TOS |
| 01011 | +2 | 10001 | R0 <- TOS-1, R1 <- TOS |
| 01011 | -1 | 00110 | R3 <- EMPTY |
| 01011 | -2 | 00000 | R2 <- EMPTY, R3 <- EMPTY |
| 01000 | +1 | 01101 | R1 <- TOS |
| 01000 | +2 | 10010 | R1 <- TOS-1, R2 <- TOS |
| 01000 | -1 | 00111 | R0 <- EMPTY |
| 01000 | -2 | 00000 | R0 <- EMPTY, R3 <- EMPTY |
| 00110 | +1 | 01011 | R3 <- TOS |
| 00110 | +2 | 01100 | R0 <- TOS, R3 <- TOS-1 |
| 00110 | +3 | 10001 | R1 <- TOS, R0 <- TOS-1, R3 <- TOS-2 |
| 00110 | -1 | 00000 | R2 <- EMPTY |
| 00111 | +1 | 01000 | R0 <- TOS |
| 00111 | +2 | 01101 | R0 <- TOS-1, R1 <- TOS |
| 00111 | +3 | 10010 | R0 <- TOS-2, R1 <- TOS-1, R2 <- TOS |
| 00111 | -1 | 00000 | R3 <- EMPTY |
| 00101 | +1 | 01010 | R2 <- TOS |
| 00101 | +2 | 01111 | R2 <- TOS-1, R3 <- TOS |
| 00101 | +3 | 10000 | R2 <- TOS-2, R3 <- TOS-1, R1 <- TOS |
| 00101 | -1 | 00000 | R1 <- EMPTY |

**TABLE 4**

It will be appreciated that the relationships between states and conditions illustrated in Table 2, Table 3 and Table 4 could be combined into a single state transition table or state diagram, but they have been shown separately above to aid clarity.

The relationships between the different states, conditions, and nett actions may be used to define a hardware state machine (in the form of a finite state machine) for controlling this aspect of the operation of the instruction translator 108. Alternatively, these relationships could be modelled by software or a combination of hardware and software.

There follows below an example of a subset of the possible Java bytecodes that indicates for each Java bytecode of the subset the associated require full, require empty and stack action values for that bytecode which may be used in conjunction with Tables 2, 3 and 4.

```
--- iconst_0

Operation:          Push int constant

Stack:              ... =>
                    ..., 0

                    Require-Full = 0
                    Require-Empty = 1
                    Stack-Action = +1

--- iadd

Operation:          Add int

Stack:              ..., value1, value2 =>
                    ..., result

                    Require-Full = 2
                    Require-Empty = 0
                    Stack-Action = -1

--- lload_0

Operation:          Load long from local variable

Stack:              ... =>
```

```
                         ..., value.word1, value.word2

                         Require-Full = 0
                         Require-Empty = 2
                         Stack-Action = +2

--- lastore

Operation:               Store into long array

Stack:                   ..., arrayref, index, value.word1, value.word2 =>
                         ...

                         Require-Full = 4
                         Require-Empty = 0
                         Stack-Action = -4

--- land

Operation                Boolean AND long

Stack:                   ..., value1.word1, value1.word2, value2.word1,
value2.word2 =>
                         ..., result.word1, result.word2

                         Require-Full = 4
                         Require-Empty = 0
                         Stack-Action = -2

--- iastore

Operation:               Store into int array

Stack:                   ..., arrayref, index, value =>
                         ...

                         Require-Full = 3
                         Require-Empty = 0
                         Stack-Action = -3

--- ineg

Operation:               Negate int

Stack:                   ..., value =>
                         ..., result

                         Require-Full = 1
                         Require-Empty = 0
                         Stack-Action = 0
```

There also follows example instruction templates for each of the Java bytecode instructions set out above. The instructions shown are the ARM instructions which implement the required behaviour of each of the Java bytecodes. The register field "TOS-3", "TOS-2", "TOS-1", "TOS", "TOS+1" and "TOS+2" may be replaced with the appropriate register specifier as read from Table 1 depending upon the mapping state currently adopted. The denotation "TOS+n" indicates the Nth register above the register currently storing the top of stack operand starting from the register storing the top of stack operand and counting upwards in register value until reaching the end of the set of registers at which point a wrap is made to the first register within the set of registers.

```
iconst_0          MOV     tos+1, #0

lload_0           LDR     tos+2, [vars, #4]
                  LDR     tos+1, [vars, #0]

iastore           LDR     Rtmp2, [tos-2, #4]
                  LDR     Rtmp1, [tos-2, #0]
                  CMP     tos-1, Rtmp2, LSR #5
                  BLXCS   Rexc
                  STR     tos, [Rtmp1, tos-1, LSL #2]

lastore           LDR     Rtmp2, [tos-3, #4]
                  LDR     Rtmp1, [tos-3, #0]
                  CMP     tos-2, Rtmp2, LSR #5
                  BLXCS   Rexc
                  STR     tos-1, [Rtmp1, tos-2, LSL #3]!
                  STR     tos, [Rtmp1, #4]

iadd              ADD     tos-1, tos-1, tos

ineg              RSB     tos, tos, #0

land              AND     tos-2, tos-2, tos
                  AND     tos-3, tos-3, tos-1
```

An example execution sequence is illustrated below of a single Java bytecode executed by a hardware translation unit 108 in accordance with the techniques described above. The execution sequence is shown in terms of an initial state progressing through a sequence of states dependent upon the instructions being executed, generating a sequence of ARM instructions as a result of the actions being performed on each state transition, the whole having the effect of translating a Java bytecode to a sequence of ARM instructions.

```
Initial state:          00000
Instruction:            iadd (Require-Full=2, Require-Empty=0, Stack-
Action=-1)
Condition:        Require-Full>0
State Transition:   00000      >0      00100
ARM Instruction(s):

Next state:       00100              LDR R0, [Rstack, #-4]!
```

```
Instruction:           iadd (Require-Full=2, Require-Empty=0, Stack-
Action=-1)
Condition:        Requite-Full>1
State Transition:   00100      >1     01000
ARM Instructions(s):
                                       LDR R3, [Rstack, #-4]!
Next state:     01000
Instruction:           iadd (Require-Full=2, Require-Empty=0, Stack-
Action=-1)
Condition:        Stack-Action=-1
State Transition:   01000      -1     00111
Instruction template:
                    ADD   tos-1, tos-1, tos
ARM Instructions(s) (after substitution):
                                       ADD R3, R3, R0
Next state:     00111
```

Figure 6 illustrates in a different way the execution of a number of further Java bytecode instructions. The top portion of Figure 6 illustrates the sequence of ARM instructions and changes of mapping states and register contents that occur upon execution of an iadd Java bytecode instruction. The initial mapping state is 00000 corresponding to all of the registers within the set of registers being empty. The first two ARM instructions generated serve to POP two stack operands into the registers storing stack operands with the top of stack "TOS" register being R0. The third ARM instruction actually performs the add operation and writes the result into register R3 (which now becomes the top of stack operand) whilst consuming the stack operand that was previously held within register R1, thus producing an overall stack action of −1.

Processing then proceeds to execution of two Java bytecodes each representing a long load of two stack operands. The require empty condition of 2 for the first Java bytecode is immediately met and accordingly two ARM LDR instructions may be issued and executed. The mapping state after execution of the first long load Java bytecode is 01101. In this state the set of registers contains only a single empty register. The next Java bytecode long load instruction has a require empty value of 2 that is not met and accordingly the first action required is a PUSH of a stack operand to the addressable memory using an ARM STR instruction. This frees up a register within the set of registers for use by a new stack operand which may then be loaded as part of the two following LDR instructions. As previously mentioned, the instruction translation may be achieved by hardware, software, or a combination of the two. Given below is a subsection of an example software interpreter generated in accordance with the above described techniques.

```
Interpret               LDRB    Rtmp, [Rjpc, #1]!
                        LDR     pc, [pc, Rtmp, lsl #2]
                        DCD     0
                        ...
                        DCD     do_iconst_0       ; Opcode 0x03
                        ...
                        DCD     do_lload_0        ; Opcode 0x1e
                        ...
                        DCD     do_iastore        ; Opcode 0x4f
                        DCD     do_lastore        ; Opcode 0x50
                        ...
                        DCD     do_iadd           ; Opcode 0x60
                        ...
                        DCD     do_ineg           ; Opcode 0x74
                        ...
                        DCD     do_land           ; Opcode 0x7f
                        ...
do_iconst_0             MOV     R0, #0
                        STR     R0, [Rstack], #4
                        B       Interpret
do_lload_0              LDMIA   Rvars, {R0, R1}
                        STMIA   Rstack!, {R0, R1}
                        B       Interpret
do_iastore              LDMDB   Rstack!, {R0, R1, R2}
                        LDR     Rtmp2, [r0, #4]
                        LDR     Rtmp1, [r0, #0]
                        CMP     R1, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     R2, [Rtmp1, R1, LSL #2]
                        B       Interpret
do_lastore              LDMDB   Rstack!, {R0, R1, R2, R3}
                        LDR     Rtmp2, [r0, #4]
                        LDR     Rtmp1, [r0, #0]
                        CMP     R1, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     R2, [Rtmp1, R1, LSL #3]!
                        STR     R3, [Rtmp1, #4]
                        B       Interpret
do_iadd                 LDMDB   Rstack!, {r0, r1}
                        ADD     r0, r0, r1
                        STR     r0, [Rstack], #4
                        B       Interpret
do_ineg                 LDR     r0, [Rstack, #-4]!
                        RSB     tos, tos, #0
                        STR     r0, [Rstack], #4
                        B       Interpret
do_land                 LDMDB   Rstack!, {r0, r1, r2, r3}
                        AND     r1, r1, r3
                        AND     r0, r0, r2
                        STMIA   Rstack!, {r0, r1}
                        B       Interpret

State_00000_Interpret   LDRB    Rtmp, [Rjpc, #1]!
                        LDR     pc, [pc, Rtmp, lsl #2]
                        DCD     0
                        ...
```

```
                            DCD     State_00000_do_iconst_0 ; Opcode 0x03
                            ...
                            DCD     State_00000_do_lload_0  ; Opcode 0x1e
                            ...
                            DCD     State_00000_do_iastore  ; Opcode 0x4f
                            DCD     State_00000_do_lastore  ; Opcode 0x50
                            ...
                            DCD     State_00000_do_iadd     ; Opcode 0x60
                            ...
                            DCD     State_00000_do_ineg     ; Opcode 0x74
                            ...
                            DCD     State_00000_do_land     ; Opcode 0x7f
                            ...
State_00000_do_iconst_0     MOV     R1, #0
                            B       State_00101_Interpret
State_00000_do_lload_0      LDMIA   Rvars, {R1, R2}
                            B       State_01010_Interpret
State_00000_do_iastore      LDMDB   Rstack!, {R0, R1, R2}
                            LDR     Rtmp2, [r0, #4]
                            LDR     Rtmp1, [r0, #0]
                            CMP     R1, Rtmp2, LSR #5
                            BCS     ArrayBoundException
                            STR     R2, [Rtmp1, R1, LSL #2]
                            B       State_00000_Interpret
State_00000_do_lastore      LDMDB   Rstack!, {R0, R1, R2, R3}
                            LDR     Rtmp2, [r0, #4]
                            LDR     Rtmp1, [r0, #0]
                            CMP     R1, Rtmp2, LSR #5
                            BCS     ArrayBoundException
                            STR     R2, [Rtmp1, R1, LSL #3]!
                            STR     R3, [Rtmp1, #4]
                            B       State_00000_Interpret
State_00000_do_iadd         LDMDB   Rstack!, {R1, R2}
                            ADD     r1, r1, r2
                            B       State_00101_Interpret
State_00000_do_ineg         LDR     r1, [Rstack, #-4]!
                            RSB     r1, r1, #0
                            B       State_00101_Interpret
State_00000_do_land         LDR     r0, [Rstack, #-4]!
                            LDMDB   Rstack!, {r1, r2, r3}
                            AND     r2, r2, r0
                            AND     r1, r1, r3
                            B       State_01010_Interpret

State_00100_Interpret       LDRB    Rtmp, [Rjpc, #1]!
                            LDR     pc, [pc, Rtmp, lsl #2]
                            DCD     0
                            ...
                            DCD     State_00100_do_iconst_0 ; Opcode 0x03
                            ...
                            DCD     State_00100_do_lload_0  ; Opcode 0x1e
                            ...
                            DCD     State_00100_do_iastore  ; Opcode 0x4f
                            DCD     State_00100_do_lastore  ; Opcode 0x50
                            ...
                            DCD     State_00100_do_iadd     ; Opcode 0x60
                            ...
                            DCD     State_00100_do_ineg     ; Opcode 0x74
                            ...
                            DCD     State_00100_do_land     ; Opcode 0x7f
                            ...
```

```
State_00100_do_iconst_0 MOV     R1, #0
                        B       State_01001_Interpret
State_00100_do_lload_0  LDMIA   Rvars, {r1, R2}
                        B       State_01110_Interpret
State_00100_do_iastore  LDMDB   Rstack!, {r2, r3}
                        LDR     Rtmp2, [r2, #4]
                        LDR     Rtmp1, [r2, #0]
                        CMP     R3, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     R0, [Rtmp1, R3, lsl #2]
                        B       State_00000_Interpret
State_00100_do_lastore  LDMDB   Rstack!, {r1, r2, r3}
                        LDR     Rtmp2, [r1, #4]
                        LDR     Rtmp1, [r1, #0]
                        CMP     r2, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     r3, [Rtmp1, r2, lsl #3]!
                        STR     r0, [Rtmp1, #4]
                        B       State_00000_Interpret
State_00100_do_iadd     LDR     r3, [Rstack, #-4]!
                        ADD     r3, r3, r0
                        B       State_00111_Interpret
State_00100_do_ineg     RSB     r0, r0, #0
                        B       State_00100_Interpret
State_00100_do_land     LDMDB   Rstack!, {r1, r2, r3}
                        AND     r2, r2, r0
                        AND     r1, r1, r3
                        B       State_01010_Interpret

State_01000_Interpret   LDRB    Rtmp, [Rjpc, #1]!
                        LDR     pc, [pc, Rtmp, lsl #2]
                        DCD     0
                        ...
                        DCD     State_01000_do_iconst_0 ; Opcode 0x03
                        ...
                        DCD     State_01000_do_lload_0  ; Opcode 0x1e
                        ...
                        DCD     State_01000_do_iastore  ; Opcode 0x4f
                        DCD     State_01000_do_lastore  ; Opcode 0x50
                        ...
                        DCD     State_01000_do_iadd     ; Opcode 0x60
                        ...
                        DCD     State_01000_do_ineg     ; Opcode 0x74
                        ...
                        DCD     State_01000_do_land     ; Opcode 0x7f
                        ...
State_01000_do_iconst_0 MOV     R1, #0
                        B       State_01101_Interpret
State_01000_do_lload_0  LDMIA   Rvars, {r1, r2}
                        B       State_10010_Interpret
State_01000_do_iastore  LDR     r1, [Rstack, #-4]!
                        LDR     Rtmp2, [R3, #4]
                        LDR     Rtmp1, [R3, #0]
                        CMP     r0, Rtmp2, LSR #5
                        BCS     ArrayBoundException
                        STR     r1, [Rtmp1, r0, lsl #2]
                        B       State_00000_Interpret
State_01000_do_lastore  LDMDB   Rstack!, {r1, r2}
                        LDR     Rtmp2, {r3, #4}
                        LDR     Rtmp1, {R3, #0}
                        CMP     r0, Rtmp2, LSR #5
```

```
                            BCS     ArrayBoundException
                            STR     r1, [Rtmp1, r0, lsl #3]!
                            STR     r2, [Rtmp1, #4]
                            B       State_00000_Interpret
State_01000_do_iadd         ADD     r3, r3, r0
                            B       State_00111_Interpret
State_01000_do_ineg         RSB     r0, r0, #0
                            B       State_01000_Interpret
State_01000_do_land         LDMDB   Rstack!, {r1, r2}
                            AND     R0, R0, R2
                            AND     R3, R3, R1
                            B       State_01000_Interpret

State_01100_Interpret       ...
State_10000_Interpret       ...
State_00101_Interpret       ...
State_01001_Interpret       ...
State_01101_Interpret       ...
State_10001_Interpret       ...
State_00110_Interpret       ...
State_01010_Interpret       ...
State_01110_Interpret       ...
State_10010_Interpret       ...
State_00111_Interpret       ...
State_01011_Interpret       ...
State_01111_Interpret       ...
State_10011_Interpret       ...
```

Figure 7 illustrates a Java bytecode instruction "laload" which has the function of reading two words of data from within a data array specified by two words of data starting at the top of stack position. The two words read from the data array then replace the two words that specified their position and to form the topmost stack entries.

In order that the "laload" instruction has sufficient register space for the temporary storage of the stack operands being fetched from the array without overwriting the input stack operands that specify the array and position within the array of the data, the Java bytecode instruction is specified as having a require empty value of 2, i.e. two of the registers within the register bank dedicated to stack operand storage must be emptied prior to executing the ARM instructions emulating the "laload" instruction. If there are not two empty registers when this Java bytecode is encountered, then store operations (STRs) may be performed to PUSH stack operands currently held within the registers out to memory so as to make space for the temporary storage necessary and meet the require empty value for the instruction.

The instruction also has a require full value of 2 as the position of the data is specified by an array location and an index within that array as two separate stack operands. The drawing illustrates the first state as already meeting the require full and require empty conditions and having a mapping state of "01001". The "laload" instruction is broken down into three ARM instructions. The first of these loads the array reference into a spare working register outside of the set of registers acting as a register cache of stack operands. The second instruction then uses this array reference in conjunction with an index value within the array to access a first array word that is written into one of the empty registers dedicated to stack operand storage.

It is significant to note that after the execution of the first two ARM instructions, the mapping state of the system is not changed and the top of stack pointer remains where it started with the registers specified as empty still being so specified.

The final instruction within the sequence of ARM instructions loads the second array word into the set of registers for storing stack operands. As this is the final instruction, if an interrupt does occur during it, then it will not be serviced until after the instruction completes and so it is safe to change the input state with this instruction by a change to the mapping state of the registers storing stack operands. In this example, the mapping state changes to "01011" which places the new top of stack pointer at the second array word and indicates that the input variables of the array reference and index value are now empty registers, i.e. marking the registers as empty is equivalent to removing the values they held from the stack.

It will be noted that whilst the overall stack action of the "laload" instruction has not changed the number of stack operands held within the registers, a mapping state swap has nevertheless occurred. The change of mapping state performed upon execution of the final operation is hardwired into the instruction translator as a function of the Java bytecode being translated and is indicated by the "swap" parameter shown as a characteristic of the "laload" instruction.

Whilst the example of this drawing is one specific instruction, it will be appreciated that the principles set out may be extended to many different Java bytecode instructions that are emulated as ARM instructions or other types of instruction.

Figure 8 is a flow diagram schematically illustrating the above technique. At step 10 a Java bytecode is fetched from memory. At step 12 the require full and require empty values for that Java bytecode are examined. If either of the require empty or require full conditions are not met, then respective PUSH and POP operations of stack operands (possibly multiple stack operands) may be performed with steps 14 and 16. It is will be noted that this particular system does not allow the require empty and require full conditions to be simultaneously unmet. Multiple passes through steps 14 and 16 may be required until the condition of step 12 is met.

At step 18, the first ARM instruction specified within the translation template for the Java bytecode concerned is selected. At step 20, a check is made as to whether or not the selected ARM instruction is the final instruction to be executed in the emulation of the Java bytecode fetched at step 10. If the ARM instruction being executed is the final instruction, then step 21 serves to update the program counter value to point to the next Java bytecode in the sequence of instructions to be executed. It will be understood that if the ARM instruction is the final instruction, then it will complete its execution irrespective of whether or not an interrupt now occurs and accordingly it is safe to update the program counter value to the next Java bytecode and restart execution from that point as the state of the system will have reached that matching normal, uninterrupted, full execution of the Java bytecode. If the test at step 20 indicates that the final bytecode has not been reached, then updating of the program counter value is bypassed.

Step 22 executes the current ARM instruction. At step 24 a test is made as to whether or not there are any more ARM instructions that require executing as part of the template. If there are more ARM instructions, then the next of these is selected at step 26 and processing is returned to step 20. If there are no more instructions, then processing proceeds to step 28 at which any mapping change/swap specified for the Java bytecode concerned is performed in order to reflect the desired top of stack location and full/empty status of the various registers holding stack operands.

Figure 8 also schematically illustrates the points at which an interrupt if asserted is serviced and then processing restarted after an interrupt. An interrupt starts to be serviced after the execution of an ARM instruction currently in progress at step 22 with whatever is the current program counter value being stored as a return point with the bytecode sequence. If the current ARM instruction executing is the final instruction within the template sequence, then step 21 will have just updated the program counter value and accordingly this will point to the next Java bytecode (or ARM instruction should an instruction set switch have just been initiated). If the currently executing ARM instruction is anything other than the final instruction in the sequence, then the program counter value will still be the same as that indicated at the start of the execution of the Java bytecode concerned and accordingly when a return is made, the whole Java bytecode will be re-executed.

Figure 9 illustrates a Java bytecode translation unit 68 that receives a stream of Java bytecodes and outputs a translated stream of ARM instructions (or corresponding control signals) to control the action of a processor core. As described previously, the Java bytecode translator 68 translates simple Java bytecodes using instruction templates into ARM instructions or sequences of ARM instructions. When each Java bytecode has been executed, then a counter value within scheduling control logic 70 is decremented. When this counter value reaches 0, then the Java bytecode translation unit 68 issues an ARM instruction branching to scheduling code that manages scheduling between threads or tasks as appropriate.

Whilst simple Java bytecodes are handled by the Java bytecode translation unit 68 itself providing high speed hardware based execution of these bytecodes, bytecodes requiring more complex processing operations are sent to a software interpreter provided in the form of a collection of interpretation routines (examples of a selection of such routines are given earlier in this description). More specifically, the Java bytecode translation unit 68 can determined that the bytecode it has received is not one which is supported by hardware translation and accordingly a branch can be made to an address dependent upon that Java bytecode where a software routine for interpreting that bytecode is found or referenced. This mechanism can also be employed when the scheduling logic 70 indicates that a scheduling operation is needed to yield a branch to the scheduling code.

Figure 10 illustrates the operation of the embodiment of Figure 9 in more detail and the split of tasks between hardware and software. All Java bytecodes are received by the Java bytecode translation unit 68 and cause the counter to be decremented at step 72. At step 74 a check is made as to whether or not the counter value has reached 0. If the counter value has reached 0 (counting down from either a predetermined value hardwired into the system or a value that may be user controlled/programmed), then a branch is made to scheduling code at step 76. Once the scheduling code has completed at step 76, control is returned to the hardware and processing proceeds to step 72, where the next Java bytecode is fetched and the counter again decremented. Since the counter reached 0, then it will now roll round to a new, non-zero value. Alternatively, a new value may be forced into the counter as part of the exiting of the scheduling process at step 76.

If the test at step 74 indicated that the counter did not equal 0, then step 78 fetches the Java bytecode. At step 80 a determination is made as to whether the fetched bytecode is a simple bytecode that may be executed by hardware translation at step 82 or requires more complex processing and accordingly should be passed out for software interpretation at step 84. If processing is passed out to software interpretation, then once this has completed control is returned to the hardware where step 72 decrements the counter again to take account of the fetching of the next Java bytecode.

Figure 11 illustrates an alternative control arrangement. At the start of processing at step 86 an instruction signal (scheduling signal) is deasserted. At step 88, a fetched Java bytecode is examined to see if it is a simple bytecode for which hardware translation is supported. If hardware translation is not supported, then control is passed out to the interpreting software at step 90 which then executes a ARM instruction routine to interpret the Java bytecode. If the bytecode is a simple one for which hardware translation is supported, then processing proceeds to step 92 at which one or more ARM instructions are issued in sequence by the Java bytecode translation unit 68 acting as a form of multi-cycle finite state machine. Once the Java bytecode has been properly executed either at step 90 or at step 92, then processing proceeds to step 94 at which the instruction signal is asserted for a short period prior to being deasserted at step 86. The assertion of the instruction signal indicates to external circuitry that an appropriate safe point has been reached at which a timer based scheduling interrupt could take place without risking a loss of data integrity due to the partial execution of an interpreted or translated instruction.

Figure 12 illustrates example circuitry that may be used to respond to the instruction signal generated in Figure 11. A timer 96 periodically generates a timer signal after expiry of a given time period. This timer signal is stored within a latch 98 until it is cleared by a clear timer interrupt signal. The output of the latch 98 is logically combined by an AND gate 100 with the instruction signal asserted at step 94. When the latch is set and the instruction signal is asserted, then an interrupt is generated as the output of the AND gate 100 and is used to trigger an interrupt that performs scheduling operations using the interrupt processing mechanisms provided within the system for standard interrupt processing. Once the interrupt signal has been generated, this in turn triggers the production of a clear timer interrupt signal that clears the latch 98 until the next timer output pulse occurs.

Figure 13 is a signal diagram illustrating the operation of the circuit of Figure 12. The processor core clock signals occur at a regular frequency. The timer 96 generates timer signals at predetermined periods to indicate that, when safe, a scheduling operation should be initiated. The timer signals are latched. Instruction signals are generated at times spaced apart by intervals that depend upon how quickly a particular Java bytecode was executed. A simple Java bytecode may execute in a single processor core clock cycle, or more typically two or three, whereas a complex Java bytecode providing a high level management type function may take several hundred processor clock cycles before its execution is completed by the software interpreter. In either case, a pending asserted latched timer signal is not acted upon to trigger a scheduling operation until the instruction signal issues indicating that it is safe for the scheduling operation to commence. The simultaneous occurrence of a latched timer signal and the instruction signal triggers the generation of an interrupt signal followed immediately thereafter by a clear signal that clears the latch 98.

Figure 18 illustrates the provision of a first subroutine 102 terminated with a return to native instruction 104. This non-native subroutine 102 is designed to be called from a native calling program 106 using an ARM branch with link instruction to pass processing to the start of the non-native subroutine 102. The non-native subroutine 102 is also provided in a second form 108 which yields the same functionality other than being terminated by return to non-native instruction 110 rather than a return to native instruction 104. This second non-native subroutine is designed so as to be called from a non-native calling program 112.

The explicit return to native instruction 104 and return to non-native instruction 110 avoid the need at the end of each subroutine 102, 108 to actively determine whether or not the calling program was native or non-native thereby speeding the overall subroutine execution by avoiding additional processing operations at its end. The arrangement of Figure 18 has the disadvantage that two non-native subroutines 102 and 108 need to be provided which consumes memory resources.

Figure 19 illustrates a refinement of the system of Figure 18. In Figure 19 a veneer non-native subroutine 114 is provided between the native calling program and the non-native subroutine. At the end of the non-native subroutine, a return to non-native instruction is always executed and either returns processing to a non-native calling program or to a non-native veneer subroutine depending upon from where it was called. The provision of a common non-native subroutine saves memory space. The function of the non-native veneer subroutine is to receive the return call from the non-native subroutine and then pass control back to the native calling program via execution of a return to native instruction. Accordingly, the non-native veneer subroutine can be small.

In order to reduce the memory resources consumed by the already small veneer non-native subroutines 114, these may be dynamically created by instructions 116 within the native caller program as they are required. The non-native veneer subroutines may be conveniently created in the stack memory of the native caller mode. The stack memory area is already provided within the system. Furthermore, the use of a stack arrangement enables the veneer non-native subroutines to be nested within one another and the nesting properly unwound as returns are made.

Figure 20 schematically illustrates a data processing system 118 for performing the data processing operations illustrated in Figure 19. The data processing system includes a processor core 120, a hardware instruction translator 122 and a main memory 124. In operation, the instruction translator 122 may provide hardware translation of Java bytecodes into ARM instructions for simple Java bytecodes with more complicated Java bytecodes being translated using a software interpreter (not illustrated). Within the main memory 124, a native instruction program 126 is executed. Immediately prior to a subroutine call to a non-native subroutine, a veneer non-native subroutine is created in the stack memory portion of the main memory 124 as pointed to by the native stack pointer. After the veneer non-native subroutine has been created, it is branched to by a native branch instruction BL<veneer>. The veneer non-native subroutine then calls the non-native subroutine 128 which performs its desired data processing functions. At the end of the non-native subroutine 128, a return to non-native instruction is executed that returns processing to the veneer non-native subroutine within the stack memory region. At the end of the veneer non-native subroutine, a return to native instruction is executed that finally returns processing to the native code 126.

Figure 21 is a flow diagram illustrating the above described operation. At step 130 native instruction code is being executed. At step 132 a non-native veneer subroutine is created on the native stack memory of the data processing system. At step 134 a branch is made to the non-native veneer subroutine created at step 132. At step 136, the veneer non-native subroutine in turn issues a branch to the main non-native subroutine. At step 138 the main non-native subroutine is executed to achieve the main data processing functions required. At step 140 a return to non-native instruction is executed at the end of the main non-native subroutine thereby returning control to the non-native veneer subroutine. At step 142, the non-native veneer subroutine terminates with a return to native instruction that passes control back to the code immediately following that already executed by step 130.

Although illustrative embodiments of the invention have been described in detail herein with reference to the accompanying drawings, it is to be understood that the invention is not limited to those precise embodiments, and that various changes and modifications can be effected therein by one skilled in the art without departing from the scope and spirit of the invention as defined by the appended claims.